An abstract machine for module replacement

An Abstract Machine for Module Replacement

Chris Walton, Dilsun Kırlı, and Stephen Gilmore

Laboratory for Foundations of Computer Science, The University of Edinburgh,
The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, Scotland, UK.

Abstract. In this paper we define an abstract machine model for the $m\lambda$ typed intermediate language. This abstract machine is used to give a formal description of the operation of run-time module replacement from the programming language Dynamic ML. The essential technical device which we employ for module replacement is a modification of two-space copying garbage collection.

1 Introduction

We have previously presented the high-level design of Dynamic ML, a variant of the Standard ML programming language which incorporates a facility for the replacement of modular components during program execution [1]. This useful facility builds upon existing compiler technology which permits the separate compilation of modular units of a Standard ML program. A suitable application problem for Dynamic ML would be the implementation of a distributed system where it is necessary to correct errors, improve run-time performance or reduce memory use, without interrupting the execution of the system.

Standard ML has a formal definition [2]. The Definition of Standard ML acts as a solid scientific platform where experiments in programming language design may be conducted. Any alteration to the Standard ML language such as ours should be investigated in the terms of the Definition. However, as readers of the Definition will know, it is silent on the topic of memory management except to say that "there are no (semantic) rules concerning disposal of inaccessible addresses" [2, page 42]. The Definition also separates the static and the dynamic semantics in such a way that the typing information inferred at compile-time is discarded before run-time. However, Dynamic ML needs some type information at run-time. These differences from Standard ML have motivated our work on a novel semantic model that would form a suitable setting for the formal definition of Dynamic ML. That model is presented in this paper.

Other authors have argued for the usefulness of a semantic model of memory management in making precise implementation notions such as memory leaks and tail recursion optimisation, developing suitable abstract machine models of memory management for this purpose [3]. Our abstract machine model for Dynamic ML serves a different purpose and this has led to the creation of a significantly different abstract machine than those used by previous authors. An essential feature of our machine is the modelling of user program exceptions, which other authors do not include.

2 A Model for Module Replacement

We introduce our first-order module-level replacement by an example to give the reader an informal understanding of its use in practice. Standard ML has interfaces called *signatures* and modules called *structures*. In our replacement model we allow the replacement of signatures by other signatures and structures by other structures, under reasonably generous conditions [1]. As our running example we consider the replacement of one implementation of a name table with another which is functionally equivalent but offers improved performance. Both implementations match the TABLE signature shown below.

```
signature TABLE = sig
    type table
    type name = string
    val empty: table
    val insert: name × table → table
    val member: name × table → bool
end;
```

We provide a facility for expressing such a replacement which ensures that the data values already present in memory cannot be used in ways which are not allowed by their type. The replacement operation is expressed by allowing the user to abstract over a Tbl structure which is specialised to implement a name table as a list of character strings. The Standard ML terminology for a structure abstraction is a *functor*. The functor body describes a structure which implements name tables as binary search trees and in addition contains functions to convert from the types of the given structure to the types of the new. We place the conversion functions inside an Install structure and follow a convention of mapping values from their old representation to their new one using functions which have the same identifier as the type which they update. This method of structure replacement is encoded as a Dynamic ML functor below.

```
functor InstallTable (Tbl: TABLE where type table = string list) :> TABLE =
struct
    type name = string
    datatype table = empty | node of table × name × table

    fun insert (s, empty) = node (empty, s, empty)
      | insert (s, node (l, v, r)) =
          if s < v then node (insert (s, l), v, r)
          else if s > v then node (l, v, insert (s, r)) else node (l, v, r)

    fun member (s, empty) = false
      | member (s, node (l, v, r)) =
          if s < v then member (s, l)
          else if s > v then member (s, r) else true

    structure Install = struct
        val name: Tbl.name → name = fn x ⇒ x
        val table: Tbl.table → table = List.foldr insert empty
    end
end;
```

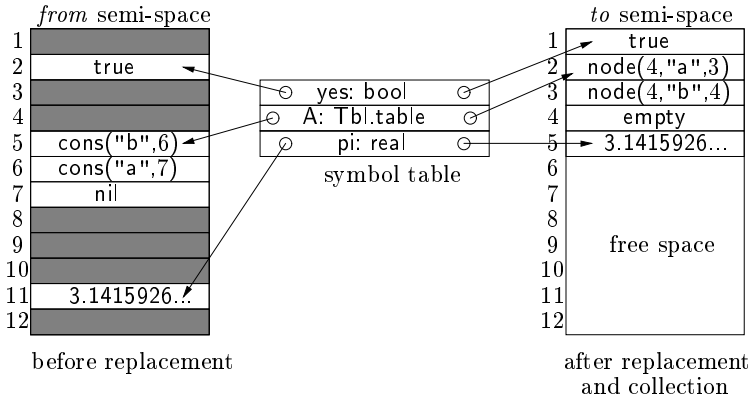

Fig. 1. Code replacement with type update

Through the use of the InstallTable functor, a Dynamic ML programmer could replace a structure which implemented tables as (either sorted or unsorted) lists with one which implemented them as binary search trees. This is an example of a very simple modification which would improve the performance of the insert and member operations. However, more sophisticated improvements would be made by the same method: defining a functor which maps the old implementation to the new one and provides functions to convert from the old types to the new. In both cases, it is critical that the types under replacement are abstract ones (with only the type identifier given in the signature) in order that functions outside the structure were not able to depend on a particular choice of concrete representation for a type, thereby preventing its replacement later.

We propose to perform the code replacement operation during garbage collection. A functor, such as the one shown above, is compiled separately. We then invoke the garbage collection operation extended with the application of the replacement functions from the Install structure to any values of the type under replacement. After completion of the copying with replacement, it is possible to dispose of the outdated version of the structure under modification (in the *from* semi-space), and switch to use the new version (in the *to* semi-space) which now contains the data values of the newly introduced replacement types. This is illustrated in Fig. 1 where a list representation of a name table containing the names b and a is replaced by the corresponding tree representation. Values of types not under replacement are unaltered: this includes values of built-in types such as booleans and real numbers.

The functions which are executed during code replacement are unrestricted Standard ML functions which may diverge upon application or raise an exception to signal an inability to continue processing. Our method of recovery is to rollback the garbage collection operation when any exception is raised. We revert to using the *from* semi-space of data values and the old types and we continue with the execution of the old program code.

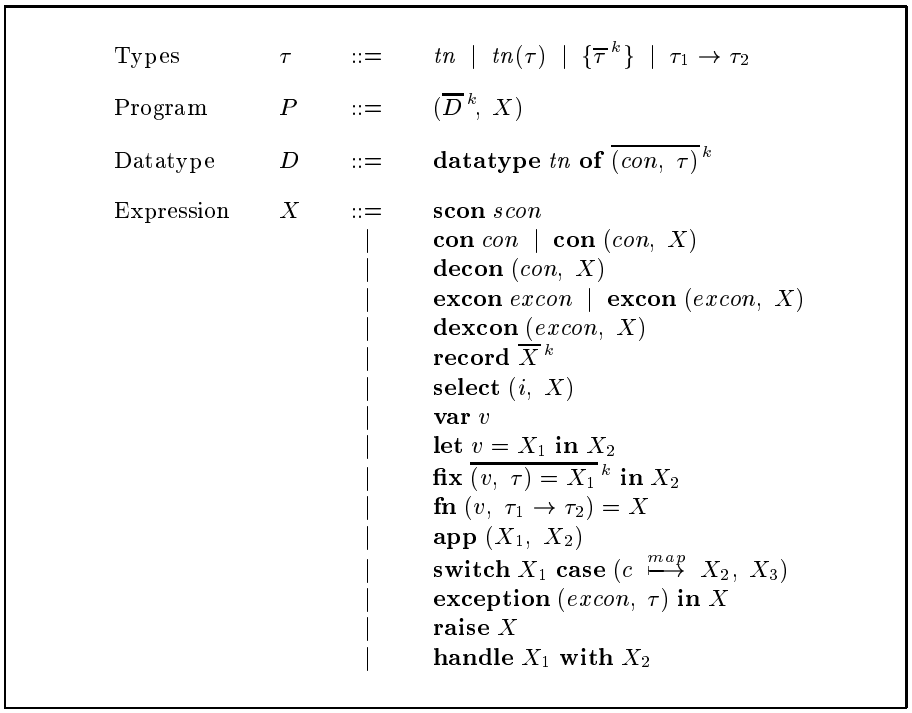

Fig. 2. Syntax of $m\lambda$ language

3 The $m\lambda$ Language

In order to formalise the replacement operation described in the previous section we first define a call-by-value lambda language $m\lambda$. This language is representative of a typical typed intermediate language used in the current state-of-the-art Standard ML compilers [4–7]. By basing replacement on such an intermediate language, we obtain an operation that is applicable to the whole of Standard ML, yet avoid a great deal of the complexity. For example, pattern matching is converted into switch statements by the higher-level match compiler. Furthermore, we can assume that the $m\lambda$ program is well-typed. For brevity, we have restricted our attention here to a purely-functional monomorphic lambda language. However, we note that including polymorphism and side-effects does not change the resulting replacement operation.

The syntax of the $m\lambda$ language is given in Fig. 2. The syntactic categories of the language include special constants of types unit, integer, real, and string; value constructors such as c_true; exception constructors such as e_match; and type names such as t_bool. Variables are bound uniquely to values generated by the evaluation of expressions. The types are constructor types (which may be either nullary or unary), record types, and function types. Constructor types

include the basic types, as required by the special constants; value constructor types; and exception constructor types.

A program consists of a sequence of datatype declarations followed by a single expression. A datatype declaration consists of a unique type name and a sequence of typed constructors. The expressions divide into those for constructing and de-constructing values, defining and manipulating variables, and controlling the order of evaluation.

Notation: A set is defined by enumerating its members in braces, for example, $\overline{\overline{x}} = \{a,\ b,\ c,\ d\}$ with $\emptyset$ for the empty set. A sequence is an ordered list of members of a set, e.g. $\overline{x}^k = (a,\ b,\ c,\ a)$. The ith element of a non-empty sequence is written x^i, where $0 < i \leq k$. A finite-map from $\overline{x}^k$ to $\overline{y}^k$ is defined: $x \stackrel{map}{\longmapsto} y = \{x^1 \mapsto y^1,\ \ldots,\ x^k \mapsto y^k\}$ (the elements of $\overline{x}^k$ must be unique). The domain (Dom) and range (Rng) are the sets of elements of $\overline{x}^k$ and $\overline{y}^k$ respectively. A stack is written as a dotted sequence, e.g. $S = (a{\cdot}b{\cdot}c)$. The left-most element of the sequence is the top of the stack, and a pair of adjacent brackets () is used to represent the empty stack.

We use the meta-variables *scon* for special constants, *con* and *excon* for value and exception constructors with c ranging over all three of these and i over special constants of integer type. We use *tn* for type names and v for variables. We use p for type heap pointers and l for value heap locations.

4 The $m\lambda$ Abstract Machine

The dynamic semantics of $m\lambda$ is formalised in this section by a transition relation between states of an abstract machine. The organisation of our abstract machine has some features in common with the $\lambda_{\mathrm{gc}}^{\rightarrow\forall}$ abstract machine [3] which is used in the formal description of the behaviour of the TIL/ML compiler. However, the resulting transitions differ considerably as $m\lambda$ is significantly different from $\lambda_{\mathrm{gc}}^{\rightarrow\forall}$. One important way in which it differs is that $m\lambda$ does not adopt the named-form representation of expressions and types.

The syntax of the abstract machine is given in Fig. 3. The state of the machine is defined by a 4-tuple $(H,\ E,\ ES,\ RS)$ of a heap, an environment, an exception stack, and a result stack. The heap is used to store all the run-time objects of the program, while the environment provides a view of the heap relevant to the fragment of the program being evaluated (for example, a mapping between the bound variables currently in scope, and their values on the heap). The exception stack stores pointers to exception handling functions (closures). The result stack holds pointers to temporary results.

The heap consists of a type-heap mapping pointers to allocated types, and a value-heap mapping locations to allocated values. The heap types correspond directly to types in the $m\lambda$ language, and the heap values correspond to the heap types. Nullary constructors *scon*, *con*, and *excon* all have type *tn*. Unary constructors $con(l)$ and $excon(l)$ have type $tn(p)$. Records $\{\overline{l}^k\}$ have type $\{\overline{p}^k\}$, and closures $\langle\langle E,\ v,\ X\rangle\rangle$ have type $p_1 \rightarrow p_2$. The type heap and value heap are

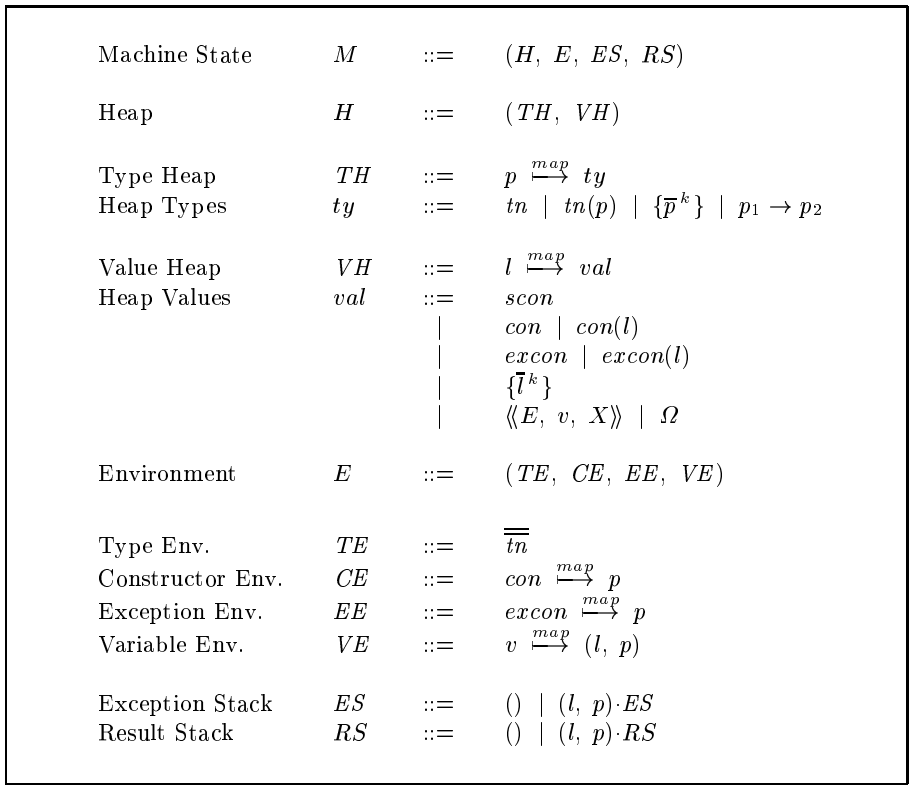

Fig. 3. Syntax of $m\lambda$ abstract machine

represented by finite-maps, as locations and pointers may be bound only once. It is important to note that we can only determine the shape of the data at a particular location by examining its corresponding type. Thus, each heap location will be paired with a heap pointer: $(l,\ p)$. This is essential for implementing tag-free garbage collection in the following section.

The following syntactic conventions are used for allocating heap objects: $H[l_1 \mapsto val_1,\ \ldots,\ l_k \mapsto val_k]$ allocates values $val_1, \ldots, val_k$ on the value heap, binding them to fresh locations $l_1, \ldots, l_k$, and $H[p_1 \mapsto \tau_1,\ \ldots,\ p_k \mapsto \tau_k]$ allocates types $\tau_1, \ldots, \tau_k$ on the type heap, binding them to fresh pointers $p_1, \ldots, p_k$. There are no corresponding operations for removing objects from the heap as this is achieved through garbage collection. However, the implementation of the fixed-point expression which is used to implement recursive functions requires a heap-update operation. As a special case, $H[l \mapsto \Omega]$ allocates a dummy closure on the value heap bound to a fresh location l. This location can subsequently be updated with a mapping to a new closure.

The environment records the allocation of $m\lambda$ values, mapping them to heap locations/pointers. As identifiers and variables are unique, their corresponding

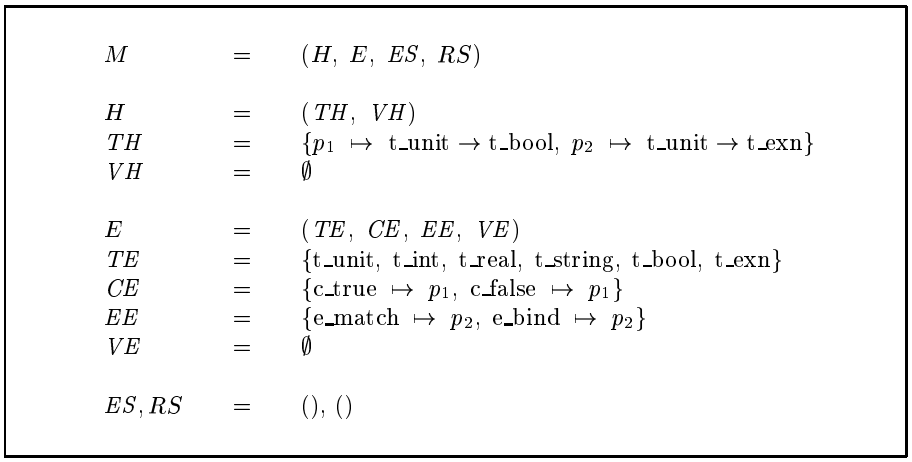

Fig. 4. Initial machine state

environments are represented by finite-maps with the exception of the type environment where it is sufficient just to use a set for type names. The following notational conventions are used for extending the environment: $E[tn]$ adds tn to the type environment, $E[con \mapsto p]$ binds the constructor con to the heap pointer p in the constructor environment. Similarly, $E[excon \mapsto p]$, and $E[v \mapsto (l,\ p)]$ denote the binding of exception constructors and lambda variables respectively to heap pointers/locations in the environment. There are no operations for removing objects from the environment. However, unlike the heap, a copy of the current environment may be made at any time, for example by creating a closure. Thus, objects can effectively be removed from the environment by reverting to an old copy of the environment.

Execution of the abstract machine is defined by a transition system between machine states. The individual transitions are listed in Appendix A. The top-level transition has the form $(H,\ E,\ ES,\ RS,\ P) \Rightarrow (H',\ E',\ ES',\ RS')$, where P is an $m\lambda$ program, $(H,\ E,\ ES,\ RS)$ is the initial machine state (as illustrated in Fig. 4), and $(H',\ E',\ ES',\ RS')$ is the final machine state. This top-level transition decomposes into a sequence of transitions of the form $(H,\ E,\ ES,\ RS,\ D) \Rightarrow (H',\ E',\ ES',\ RS')$ for processing the datatypes D, followed by a sequence $(H,\ E,\ ES,\ RS,\ X) \Rightarrow (H',\ E',\ ES',\ RS')$ for evaluating the expression X.

There are three possible outcomes which can result from evaluating this expression. Firstly, the sequence may terminate normally yielding a single pair $(l,\ p)$ in the result stack which references the result. Secondly, the sequence may terminate prematurely, through an uncaught exception, yielding a pair $(l,\ p)$ at the top of the result stack which references the exception. Thirdly, the machine may encounter an infinite sequence of transitions and fail to terminate.

5 Garbage Collection with Replacement

In Section 2 we have explained how we extend the traditional two-space copying garbage collection to implement our replacement operation. In this section, we give the formal definition of this extended garbage collection used in the abstract machine defined in Section 4. The replacement operation has been presented in terms of the use of the modular constructs of Standard ML. However, for brevity we restrict our discussion here to the simpler non-modular language presented in Section 3.

We will consider the case where we are equipped with the information represented by a semantic object defined as follows:

$$RM ::= p_{\text{old}} \overset{map}{\longmapsto} (l_{\text{rep}},\ p_{\text{rep}})$$

The domain of the replacement map $Dom(RM)$ is the set of the pointers to the types that are to be dynamically replaced. Each element p_{old} of the domain is mapped to a location/type-pointer pair $(l_{\text{rep}},\ p_{\text{rep}})$. The location contains the closure of the function which is to execute the replacement operation and the type-pointer points to the type which is to replace the old type.

In Dynamic ML this information is extracted from the result of the evaluation of the sub-structure Install which contains the user defined functions dedicated to the replacement operation. The replacement map obtained from the Install structure of our example would be as follows:

$$\{p_{Tbl.name} \ \mapsto\ (l_{name}, p_{name}),\quad p_{Tbl.table} \ \mapsto\ (l_{table}, p_{table})\}$$

We define garbage as the objects that are not reachable either directly or indirectly from the environment, exception stack, or result stack. Garbage collection may take place before or after any transition of the $m\lambda$ abstract machine dropping the bindings of the unreachable objects provided that this does not change the observable behaviour of the program.

Garbage collection is defined as a rewriting system between the configurations of our abstract machine $(S,\ RM,\ H_f,\ H_t)$. The replacement map denoted by RM is the auxiliary data structure which provides the information necessary for the replacement operation. The traditional two-space copying garbage collection corresponds to the case where RM is empty.

Initially, the scan stack S contains all of the pointers p and $(l,\ p)$ pairs in E, ES, and RS. Heap objects are copied from the semi-space H_f to the semi-space H_t until the scan stack is empty according to the rules listed in Appendix B.

We can incorporate the garbage collection operation in the dynamic semantics of our language explicitly by means of the following evaluation rule:

$$\frac{(ES{\cdot}RS{\cdot}FE(E),\ RM,\ H_f,\ \emptyset) \Rightarrow^{*}_{\text{gc}} ((),\ \emptyset,\ H_f,\ H_t) \quad (H_t,\ E,\ ES,\ RS,\ X)\ \Rightarrow\ (H_1,\ E_1,\ ES_1,\ RS_1)}{(H_f,\ E,\ ES,\ RS,\ X)\ \Rightarrow\ (H_1,\ E_1,\ ES_1,\ RS_1)}$$

where $\Rightarrow_{\mathrm{gc}}^{*}$ stands for the repeated application of the $\Rightarrow_{\mathrm{gc}}$ rules. The informal understanding of the $\Rightarrow_{\mathrm{gc}}$ rules is as follows:

Rule R0 is applied when the scan stack is empty. This signals the end of the garbage collection operation. The replacement map is discarded in order for subsequent garbage collections to operate correctly.

Rules R1, R1† and R1‡ are applied when the top of the scan stack is a location/type-pointer pair $(l,\ p)$ and the value in the location has not yet been copied to the H_t semi-space, i.e. $l \notin Dom(H_t)$.

In R1 the type of the value reveals that it need not be replaced. As a result, the value in the H_f semi-space is copied to the H_t semi-space. The free locations and the type pointers of the allocated value are added to the scan stack.

R1† and R1‡ are variants of R1 where the type of the value indicates that the value is to be replaced i.e. $p \in Dom(RM)$. Consecutive lookups in the replacement map and the heap yield the closure of the replacement function that is to be applied to the value currently being scanned. The code of the closure is evaluated in the environment extended by the binding of the scanned value. Note also that the disjoint union of the two semi-spaces is assumed as the heap because the code may be referring to some location or type-pointer that has already been copied.

There are two possible outcomes for the garbage collection operation. Either evaluation ends successfully or an exception is raised by one of the functions which is updating the values from the old type to the new one. These two cases are distinguished by inspecting the type of the most recent result which is at the top of the result stack. The first case is captured by R1†. The new value is copied to the H_t semi-space and the scan stack is arranged as in R1. The second case is captured by R1‡ where the top of the stack indicates that a top level exception has been raised. According to our implementation model we rollback the garbage collection operation and revert to using the H_f semi-space values. This is indicated by setting the scan stack to empty and identifying H_t with H_f. The replacement map is discarded as in R0.

R2 is applied when the top of the scan stack is a location/type-pointer pair and the value in the location has already been copied to the H_t semi-space. It simply skips this location and continues with the rest of the scan stack. R4 is exactly like R2 but skips over a type pointer instead of a location.

R3 and R3† are applied when the top of the scan stack is a type pointer and the type pointer has not yet been copied to the H_t semi-space. R3 deals with the case where the type need not be replaced. The free pointers of the allocated type are added to the scan stack and the old representation of the type is copied to the H_t semi-space. R3† deals with the case where the old representation of the type is to be replaced by the new representation.

The functions FE, FP and FL employed in the rewriting rules compute the free location/type-pointer pairs $(l,\ p)$ and type-pointers p. They are given in Fig. 5.

$$FE(E) = Rng(CE) \cdot Rng(EE) \cdot Rng(VE)$$

$$\begin{aligned} FP(tn) &= () \\ FP(tn(p)) &= (p) \\ FP(\{\overline{p}^k\}) &= (p^1 \cdots p^k) \\ FP(p_1 \to p_2) &= FP(p_1) \cdot FP(p_2) \end{aligned}$$

$$\begin{aligned} &FL(H,\ l,\ tn) = () \\ &FL(H,\ l_1,\ tn(p)) = (l_2,\ p) && \text{where } tn = \text{t\_exn and } H(l_1) = excon(l_2) \\ &FL(H,\ l_1,\ tn(p)) = (l_2,\ p) && \text{where } tn \neq \text{t\_exn and } H(l_1) = con(l_2) \\ &FL(H,\ l,\ \{\overline{p}^k\}) = (l^1,\ p^1) \cdots (l^k,\ p^k) && \text{where } H(l) = \{\overline{l}^k\} \\ &FL(H,\ l,\ p_1 \to p_2) = FE(E) && \text{where } H(l) = \langle\!\langle E,\ v,\ X \rangle\!\rangle \end{aligned}$$

Fig. 5. Auxiliary functions for garbage collection

6 Practicality

Users of state-of-the-art compilers for modern programming languages have become accustomed to complex program analyses which safely deliver impressive performance benefits in terms of run-time and memory usage while simultaneously offering greater access to a more sophisticated model of computation which incorporates advanced features such as remote evaluation or code mobility. In this setting it is all too easy to invent a new paradigm for program execution and to claim that it can be implemented efficiently because modern compilers and run-time systems offer so much functionality and convenience. In this section we would like to provide a more concrete explanation of the key implementation technology which could be used to provide an efficient implementation of the code replacement operation which we have described.

Languages in the Standard ML family are strongly typed. In order to enforce the application of the type-checking stage these language make a strict distinction between *elaboration* and *evaluation*, insisting that programs which have not successfully elaborated cannot be evaluated at all. The rigid ordering of these two stages prohibits the execution of any programs which attempt to use data values in ways which are not allowed by their type and thus eliminates a large number of software errors which would manifest themselves at run-time if working in an untyped programming language. However, several authors have observed that two stages are not enough for complex applications such as program generators. This has led to approaches such as the *multi-stage* programming paradigm for MetaML [8], *staged type inference* [9] and the *staged compilation* paradigm for the language Modal ML [10]. The last of these is the most closely related to our own approach because it has demonstrated the effectiveness of the use of run-time code generation by Lee and colleagues in the development of the Fabius compiler for ML [11]. Using this technology it is possible for us to eliminate

the run-time penalties incurred by the use of abstract types in module specifications by exploiting the underlying representation of an abstract type and re-compiling at run-time when the replacement module is available. Further, many other benefits come from the use of run-time code generation including those associated with *partial evaluation* [12] since it is possible to take advantage of values which are not known until run-time. Other standard compiler optimisations such as elimination of array-bounds checking and loop unrolling also become more profitable in this setting.

Our discussion of module replacement has been exclusively framed in the context of Standard ML but the same idea has recently been investigated by other authors working with other languages. Andersson and colleagues [13] have considered the dynamic replacement of loaded classes in the Java run-time system. Their approach to implementation differs from ours in that they perform replacement of objects of the outdated class as they are accessed, meaning that both versions of the class are active at the same time. Replaced objects are garbage-collected as the computation proceeds and whenever all of the objects of the old version of the class have been replaced the class object will have no more references and it can then be garbage-collected also.

It might seem that the idea of dynamic replacement is better suited to an embedded systems language for a system with a high availability requirement, making Java the better choice for investigating dynamic code replacement because that its intended application domain. Although we admire Java as a useful, soundly-engineered product the absence of a well-understood theory for the language makes it less well-suited for this issue. Other researchers are also considering the use of Standard ML in areas such as these [14].

7 Conclusions

Modern compilers for higher-order typed programming languages use typed intermediate languages to structure the compilation process. We have provided an abstract machine definition of a small functional language which is representative of these. This has allowed us to define precisely the operation of dynamic module replacement which is used in Dynamic ML.

In composing the Definition of Standard ML, the authors chose not to give an account of the operation of garbage collection, which most compilers for that language provide. This was the right decision when focusing upon the abstract description of a sophisticated high-level language such as Standard ML. Our concern was to describe part of the operation of an executing computation, with access to values described by concrete manifestations.

The use of an abstract machine notation has allowed us to isolate the novel feature of interest from our language. We have presented its definition separately from other aspects such as syntax and type-correctness. For our purposes, the use of an abstract machine has established the right level of detail. In addition, it provides an implementor with an unambiguous and precise description of the operation of module-level code replacement.

Acknowledgements

Dilsun Kırlı is supported by a University of Edinburgh scholarship from the Department of Computer Science. Chris Walton is supported by an EPSRC postgraduate studentship.

References

1. S. Gilmore, D. Kırlı, and C. Walton. Dynamic ML without Dynamic Types. Technical Report ECS-LFCS-97-378, Laboratory for Foundations of Computer Science, The University of Edinburgh, 1997.
2. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML: Revised 1997*. The MIT Press, 1997.
3. G. Morrisett and R. Harper. Semantics of Memory Management for Polymorphic Languages. Technical report, School of Computer Science, Carnegie Mellon University, 1996. Also published as Fox Memorandum CMU-CS-FOX-96-04.
4. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimising compiler for ML. In *ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, 1996.
5. Z. Shao. An Overview of the FLINT/ML Compiler. Technical report, Department of Computer Science, Yale University, 1997.
6. M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. Olesen, P. Sestoft, and P. Bertelsen. Programming with Regions in the ML-Kit. Technical Report DIKU-TR-97/12, Department of Computer Science, University of Copenhagen, 1997.
7. N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Third ACM SIGPLAN International Conference on Functional Programming*, Baltimore, 1998.
8. W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, Amsterdam, The Netherlands, June 1997.
9. M. Shields, T. Sheard, and S. Peyton Jones. Dynamic typing as staged type inference. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, January 1998.
10. P. Wickline, P. Lee, and F. Pfenning. Run-time code generation and Modal-ML. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 224–235, Montreal, Canada, June 1998.
11. P. Lee and M. Leone. Optimising ML with run-time code generation. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 137–148, Philadelphia, Pennsylvania, May 1996.
12. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
13. J. Andersson, M. Comstedt, and T. Ritzau. Run-time support for dynamic Java architectures. In *ECOOP'98 Workshop on Object-Oriented Software Architectures*, Brussels, July 1998.
14. R. Pucella. Reactive programming in Standard ML. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 48–57, Chicago, USA, May 1998. IEEE Computer Society Press.

A Abstract Machine Definition

A.1 Programs

$$\frac{\begin{array}{l}P = (\overline{D}^{\,k},\ X) \\ (H,\ E,\ ES,\ RS,\ D^1) \Rightarrow (H_1,\ E_1,\ ES,\ RS)\ \ldots \\ \ldots\ (H_{k-1},\ E_{k-1},\ ES,\ RS,\ D^k) \Rightarrow (H_k,\ E_k,\ ES,\ RS) \\ (H_k,\ E_k,\ ES,\ RS,\ X) \Rightarrow (H_{k+1},\ E_{k+1},\ ES_1,\ RS_1)\end{array}}{(H,\ E,\ ES,\ RS,\ P) \Rightarrow (H_{k+1},\ E_{k+1},\ ES_1,\ RS_1)}$$

A.2 Datatypes

$$\frac{}{\begin{array}{l}(H,\ E,\ ES,\ RS,\ \textbf{datatype}\ tn\ \textbf{of}\ \overline{(con,\ \tau)}^{\,k}) \Rightarrow \\ \quad (H[p_1\ \mapsto\ \tau^1 \rightarrow tn,\ \ldots,\ p_k\ \mapsto\ \tau^k \rightarrow tn], \\ \quad\ E[tn][con^1\ \mapsto\ p_1,\ \ldots,\ con^k\ \mapsto\ p_k],\ ES,\ RS)\end{array}}$$

A.3 Expressions

$$\frac{}{(H,\ E,\ ES,\ RS,\ \textbf{scon}\ scon) \Rightarrow (H[l\ \mapsto\ scon][p\ \mapsto\ \tau_{scon}],\ E,\ ES,\ (l,\ p)\cdot RS)}$$

$$\frac{E(con) = p_1 \qquad H(p_1) = p_2 \rightarrow p_3}{(H,\ E,\ ES,\ RS,\ \textbf{con}\ con) \Rightarrow (H[l_1\ \mapsto\ con],\ E,\ ES,\ (l_1,\ p_3)\cdot RS)}$$

$$\frac{\begin{array}{l}(H,\ E,\ ES,\ RS,\ X) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1)\cdot RS) \\ E(con) = p_2 \qquad H_1(p_2) = p_3 \rightarrow p_4\end{array}}{(H,\ E,\ ES,\ RS,\ \textbf{con}\ (con,\ X)) \Rightarrow (H_1[l_2\ \mapsto\ con(l_1)],\ E,\ ES,\ (l_2,\ p_4)\cdot RS)}$$

$$\frac{\begin{array}{l}(H,\ E,\ ES,\ RS,\ X) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1)\cdot RS) \\ E(con) = p_2 \qquad H_1(p_2) = p_3 \rightarrow p_4 \qquad H_1(l_1) = con(l_2)\end{array}}{(H,\ E,\ ES,\ RS,\ \textbf{decon}\ (con,\ X)) \Rightarrow (H_1,\ E,\ ES,\ (l_2,\ p_3)\cdot RS)}$$

$$\frac{E(excon) = p_1 \qquad H(p_1) = p_2 \rightarrow p_3}{(H,\ E,\ ES,\ RS,\ \textbf{excon}\ excon) \Rightarrow (H[l_1\ \mapsto\ excon],\ E,\ ES,\ (l_1,\ p_3)\cdot RS)}$$

$$\frac{\begin{array}{l}(H,\ E,\ ES,\ RS,\ X) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1)\cdot RS) \\ E(excon) = p_2 \qquad H_1(p_2) = p_3 \rightarrow p_4\end{array}}{\begin{array}{l}(H,\ E,\ ES,\ RS,\ \textbf{excon}\ (excon,\ X)) \Rightarrow \\ \quad (H_1[l_2\ \mapsto\ excon(l_1)],\ E,\ ES,\ (l_2,\ p_4)\cdot RS)\end{array}}$$

$$\frac{\begin{array}{l}(H,\ E,\ ES,\ RS,\ X) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1)\cdot RS) \\ E(excon) = p_2 \qquad H_1(p_2) = p_3 \rightarrow p_4 \qquad H_1(l_1) = excon(l_2)\end{array}}{(H,\ E,\ ES,\ RS,\ \textbf{dexcon}\ (excon,\ X)) \Rightarrow (H_1,\ E,\ ES,\ (l_2,\ p_3)\cdot RS)}$$

$$\frac{\begin{array}{l}(H,\ E,\ ES,\ RS,\ X^1) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1){\cdot}RS)\ \ldots \\ \ldots\ (H_{k-1},\ E,\ ES,\ (l_{k-1},\ p_{k-1}) \cdots (l_1,\ p_1){\cdot}RS,\ X^k) \Rightarrow \\ \quad (H_k,\ E,\ ES,\ (l_k,\ p_k) \cdots (l_1,\ p_1){\cdot}RS)\end{array}}{\begin{array}{l}(H,\ E,\ ES,\ RS,\ \mathbf{record}\ \overline{X}^k) \Rightarrow \\ \quad (H[l \mapsto \{l_1,\ \ldots,\ l_k\}][p \mapsto \{p_1,\ \ldots,\ p_k\}],\ E,\ ES,\ (l,\ p){\cdot}RS)\end{array}}$$

$$\frac{\begin{array}{l}(H,\ E,\ ES,\ RS,\ X) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1){\cdot}RS) \\ H_1(l_1) = \{\overline{l}^k\} \qquad H_1(p_1) = \{\overline{p}^k\}\end{array}}{(H,\ E,\ ES,\ RS,\ \mathbf{select}\ (i,\ X)) \Rightarrow (H_1,\ E,\ ES,\ (l^i,\ p^i){\cdot}RS)}$$

$$\frac{E(v) = (l,\ p)}{(H,\ E,\ ES,\ RS,\ \mathbf{var}\ v) \Rightarrow (H,\ E,\ ES,\ (l,\ p){\cdot}RS)}$$

$$\frac{\begin{array}{l}(H,\ E,\ ES,\ RS,\ X_1) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1){\cdot}RS) \\ (H_1,\ E[v \mapsto (l_1,\ p_1)],\ ES,\ RS,\ X_2) \Rightarrow (H_2,\ E_2,\ ES,\ RS_2)\end{array}}{(H,\ E,\ ES,\ RS,\ \mathbf{let}\ v = X_1\ \mathbf{in}\ X_2) \Rightarrow (H_2,\ E,\ ES,\ RS_2)}$$

$$\frac{\begin{array}{l}(H[l_f^1 \mapsto \Omega,\ \ldots,\ l_f^k \mapsto \Omega][p_f^1 \mapsto \tau^1,\ \ldots,\ p_f^k \mapsto \tau^k],\ ES,\ RS,\ X_1^1) \Rightarrow \\ \quad (H_1,\ E_1,\ ES,\ (l_1,\ p_1){\cdot}RS) \\ (H_1[l_f^1 \stackrel{upd}{\longmapsto} l_1],\ E_1,\ ES,\ RS,\ X_1^2) \Rightarrow (H_2,\ E_1,\ ES,\ (l_2,\ p_2){\cdot}RS)\ \ldots \\ \ldots\ (H_{k-1}[l_f^{k-1} \stackrel{upd}{\longmapsto} l_{k-1}],\ E_1,\ ES,\ RS,\ X_1^k) \Rightarrow (H_k,\ E_1,\ ES,\ (l_k,\ p_k){\cdot}RS) \\ (H_k[l_f^k \stackrel{upd}{\longmapsto} l_k],\ E_1,\ ES,\ RS,\ X_2) \Rightarrow (H_{k+1},\ E_1,\ ES,\ RS_{k+1})\end{array}}{(H,\ E,\ ES,\ RS,\ \mathbf{fix}\ (v,\ \tau) = X_1^{\,k}\ \mathbf{in}\ X_2) \Rightarrow (H_{k+1},\ E,\ ES,\ RS_{k+1})}$$

$$\frac{}{\begin{array}{l}(H,\ E,\ ES,\ RS,\ \mathbf{fn}\ (v,\ \tau_1 \to \tau_2) = X) \Rightarrow \\ \quad (H[l \mapsto \langle\!\langle E,\ v,\ X \rangle\!\rangle][p \mapsto \tau_1 \to \tau_2],\ E,\ ES,\ (l,\ p){\cdot}RS)\end{array}}$$

$$\frac{\begin{array}{l}(H,\ E,\ ES,\ RS,\ X_1) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1){\cdot}RS) \\ (H_1,\ E,\ ES,\ (l_1,\ p_1){\cdot}RS,\ X_2) \Rightarrow (H_2,\ E,\ ES,\ (l_2,\ p_2){\cdot}(l_1,\ p_1){\cdot}RS) \\ H_2(l_1) = \langle\!\langle E_1,\ v,\ X_3 \rangle\!\rangle \\ (H_2,\ E_1[v \mapsto (l_2,\ p_2)],\ ES,\ RS,\ X_3) \Rightarrow (H_3,\ E_2,\ ES,\ RS_3)\end{array}}{(H,\ E,\ ES,\ RS,\ \mathbf{app}\ (X_1,\ X_2)) \Rightarrow (H_3,\ E,\ ES,\ RS_3)}$$

$$\frac{\begin{array}{l}(H,\ E,\ ES,\ RS,\ X_1) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1){\cdot}RS) \\ cmap = c \stackrel{map}{\longmapsto} X_2 \qquad H(l_1) = val \\ X_4 = \text{if } val \in Dom(cmap) \text{ then } cmap(val) \text{ else } X_3 \\ (H_1,\ E,\ ES,\ RS,\ X_4) \Rightarrow (H_2,\ E,\ ES,\ RS_2)\end{array}}{(H,\ E,\ ES,\ RS,\ \mathbf{switch}\ X_1\ \mathbf{case}\ (cmap,\ X_3)) \Rightarrow (H_2,\ E,\ ES,\ RS_2)}$$

$$\frac{(H[p \mapsto \tau],\ E[excon \mapsto p],\ ES,\ RS,\ X) \Rightarrow (H_1,\ E_1,\ ES,\ RS_1)}{(H,\ E,\ ES,\ RS,\ \mathbf{exception}\ (excon,\ \tau)\ \mathbf{in}\ X) \Rightarrow (H_1,\ E,\ ES,\ RS_1)}$$

$$\frac{(H,\ E,\ (),\ RS,\ X) \Rightarrow (H_1,\ E,\ (),\ RS_1)}{(H,\ E,\ (),\ RS,\ \mathbf{raise}\ X) \Rightarrow (H_1,\ E,\ (),\ RS_1)}$$

$$\frac{(H,\ E,\ (l_1,\ p_1)\cdot ES,\ RS,\ X) \Rightarrow (H_1,\ E,\ (l_1,\ p_1)\cdot ES,\ RS_1) \quad H_1(l_1) = \langle\!\langle E_1,\ v,\ X_1 \rangle\!\rangle \quad (H_1,\ E_1[v \mapsto (l_1,\ p_1)],\ ES,\ RS,\ X_1) \Rightarrow (H_2,\ E_2,\ ES,\ RS_2)}{(H,\ E,\ (l_1,\ p_1)\cdot ES,\ RS,\ \mathbf{raise}\ X) \Rightarrow (H_2,\ E,\ ES,\ RS_2)}$$

$$\frac{(H,\ E,\ ES,\ RS,\ X_1) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1)\cdot RS) \quad (H_1,\ E,\ (l_1,\ p_1)\cdot ES,\ RS,\ X_2) \Rightarrow (H_2,\ E,\ ES_2,\ RS_2)}{(H,\ E,\ ES,\ RS,\ \mathbf{handle}\ X_1\ \mathbf{with}\ X_2) \Rightarrow (H_2,\ E,\ ES,\ RS_2)}$$

B Garbage Collection with Replacement

$$\frac{}{((),\ RM,\ H_f,\ H_t) \Rightarrow_{\mathrm{gc}} ((),\ \emptyset,\ H_f,\ H_t)} \tag{R0}$$

$$\frac{l \notin Dom(H_t) \qquad p \notin Dom(RM) \qquad H_f(l) = val}{((l,\ p)\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathrm{gc}} (p\cdot FL(H_f,\ l,\ p)\cdot S,\ RM,\ H_f,\ H_t[l \mapsto val])} \tag{R1}$$

$$\frac{\begin{array}{l} l \notin Dom(H_t) \qquad p \in Dom(RM) \\ RM(p) = (l_{\mathrm{rep}},\ p_{\mathrm{rep}}) \qquad H(l_{\mathrm{rep}}) = \langle\!\langle E_1,\ v,\ X \rangle\!\rangle \\ (H_f \uplus H_t,\ E_1[v \mapsto (l,\ p)],\ ES,\ RS,\ X) \Rightarrow (H_2,\ E_2,\ ES,\ (l_{\mathrm{new}},\ p_{\mathrm{new}})\cdot RS) \\ H_2(l_{\mathrm{new}}) = val \qquad H_2(p_{\mathrm{new}}) \neq \mathrm{t\_exn} \end{array}}{((l,\ p)\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathrm{gc}} (p\cdot FL(H_f,\ l,\ p)\cdot S,\ RM,\ H_f,\ H_t[l \mapsto val])} \tag{R1†}$$

$$\frac{\begin{array}{l} l \notin Dom(H_t) \qquad p \in Dom(RM) \\ RM(p) = (l_{\mathrm{rep}},\ p_{\mathrm{new}}) \qquad H(l_{\mathrm{rep}}) = \langle\!\langle E_1,\ v,\ X \rangle\!\rangle \\ (H_f \uplus H_t,\ E_1[v \mapsto (l,\ p)],\ ES,\ RS,\ X) \Rightarrow (H_2,\ E_2,\ ES,\ (l_{\mathrm{new}},\ p_{\mathrm{new}})\cdot RS) \\ H_2(p_{\mathrm{new}}) = \mathrm{t\_exn} \end{array}}{((l,\ p)\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathrm{gc}} ((),\ \emptyset,\ H_f,\ H_f)} \tag{R1‡}$$

$$\frac{l \in Dom(H_t)}{((l,\ p)\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathrm{gc}} (S,\ RM,\ H_f,\ H_t)} \tag{R2}$$

$$\frac{p \notin Dom(H_t) \qquad p \notin Dom(RM) \qquad H_f(p) = ty}{(p\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathrm{gc}} (FP(ty)\cdot S,\ RM,\ H_f,\ H_t[p \mapsto ty])} \tag{R3}$$

$$\frac{\begin{array}{l} p \notin Dom(H_t) \qquad p \in Dom(RM) \\ RM(p) = (l_{\mathrm{rep}},\ p_{\mathrm{rep}}) \qquad H_f(p_{\mathrm{rep}}) = ty \end{array}}{(p\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathrm{gc}} (FP(ty)\cdot S,\ RM,\ H_f,\ H_t[p \mapsto ty])} \tag{R3†}$$

$$\frac{p \in Dom(H_t)}{(p\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathrm{gc}} (S,\ RM,\ H_f,\ H_t)} \tag{R4}$$